Yet Another Compressed Cache: a Low Cost Yet Effective Compressed Cache

Somayeh Sardashti, André Seznec, David A. Wood

Yet Another Compressed Cache:

a Low Cost Yet Effective Compressed Cache

SOMAYEH SARDASHTI, University of Wisconsin-Madison
ANDRE SEZNEC, IRISA/INRIA
DAVID A WOOD, University of Wisconsin-Madison

Cache memories play a critical role in bridging the latency, bandwidth, and energy gaps between cores and off-chip memory. However, caches frequently consume a significant fraction of a multicore chip's area, and thus account for a significant fraction of its cost. Compression has the potential to improve the effective capacity of a cache, providing the performance and energy benefits of a larger cache while using less area. The design of a compressed cache must address two important issues: i) a low-latency, low-overhead compression algorithm that can represent a fixed-size cache block using fewer bits and ii) a cache organization that can efficiently store the resulting variable-size compressed blocks. This paper focuses on the latter issue.

In this paper, we propose YACC (Yet Another Compressed Cache), a new compressed cache design that targets improving effective cache capacity with a simple design. YACC uses super-blocks to reduce tag overheads, while packing variable-size compressed blocks to reduce internal fragmentation. YACC achieves the benefits of two state-of-the art compressed caches, Decoupled Compressed Cache (DCC) [Sardashti et al. 2013] and Skewed Compressed Cache (SCC) [Sardashti et al. 2014], with a more practical and simpler design. YACC's cache layout is similar to conventional caches, with a largely unmodified tag array and unmodified data array. Compared to DCC and SCC, YACC requires neither the significant extra metadata (i.e., back-pointers) needed by DCC to track blocks nor the complexity and overhead of skewed associativity (i.e., indexing ways differently) needed by SCC. An additional advantage over previous work is that YACC enables modern replacement mechanisms, such as RRIP.

For our benchmark set, compared to a conventional uncompressed 8MB LLC, YACC improves performance by 8% on average and up to 26%, and reduces total energy by on average 6% and up to 20%. An 8MB YACC achieves approximately the same performance and energy improvements as a 16MB conventional cache at a much smaller silicon footprint, with only 1.6% greater area than an 8MB conventional cache. YACC performs comparably to DCC and SCC, but is much simpler to implement.

• Computer systems organization → Architectures → Multicore architectures

Additional Key Words and Phrases: compression, cache design, energy efficiency, performance, multi-core systems

ACM Reference Format:

Somayeh Sardashti, Andre Seznec, and David A. Wood, 2016. Yet Another Compressed Cache: a Low Cost Yet Effective Compressed Cache. *ACM Transactions on Architecture and Code Optimization.*

This work is supported in part by the European Research Council Advanced Grant DAL No 267175 and the National Science Foundation (CNS-1117280, CCF-1218323, CNS-1302260, CCF-1438992, and CCF-1533885). The views expressed herein are not necessarily those of the NSF. Professor Wood has significant financial interests in AMD, Google, and Panasas. The authors would like to acknowledge Hamid Reza Ghasemi, members of the Multifacet research group, and our anonymous reviewers for their helpful comments.

1. INTRODUCTION

Cache memories play an increasingly critical role in bridging the latency, bandwidth, and energy gaps between cores and main memory. However, caches frequently consume a significant fraction of a multicore chip's area, and thus account for a significant fraction of its cost. Compression has the potential to improve the effective capacity of a cache, providing the performance and energy benefits of a larger cache while using less area. Prior work has proposed compression algorithms suitable for last-level cache (LLC) designs, with low latencies and efficient hardware implementations [Alameldeen and Wood 2004; Ziv et al. 1977; Ziv et al. 1978; Huffman. 1952; Vitter. 1987; Pekhimenko et al. 2012]. Other work has focused on the organization of a compressed cache, which must efficiently compact and retrieve variable-size compressed cache blocks [Alameldeen and Wood 2004; Hallnor, and Reinhardt. 2005; Kim et al. 2002; Sardashti and Wood 2013; Sardashti et al. 2014].

A compressed cache organization must provide tags to map additional blocks, support variable-size data allocation, and maintain the mappings between tags and data. The tag array must allow a compressed cache to hold more compressed than uncompressed blocks, but without incurring large area overheads. Variable-size data allocation allows compressed blocks to be stored in the cache with low internal fragmentation (i.e., low wasted space), but may require expensive re-compaction when a block changes and requires more space. In addition, the combination of additional tags and variable-size blocks makes it challenging to maintain the mapping between tags and data. Finally, a compressed cache would ideally enable advanced LLC replacement policies, such as RRIP [Jaleel et al. 2010].

State of the art compressed caches, Decoupled Compressed Cache (DCC) [Sardashti et al. 2013] and Skewed Compressed Cache (SCC) [Sardashti et al. 2014], try to achieve some of these goals, but at extra costs and complexities. Decoupled Compressed Cache (DCC) [Sardashti and Wood 2013] proposes tracking compressed blocks at super-block level to reduce tag overhead. DCC uses super-block tags where one tag tracks up to 4 neighboring blocks. DCC compresses each 64-byte block into zero to four 16-byte sub-blocks and uses a decoupled tag-data mapping to allow them to be stored anywhere in a cache set. This flexible sub-block allocation eliminates re-compaction overheads when a block size grows, but requires the additional area and complexity of backward pointers to maintain the decoupled mapping.

Skewed Compressed Cache (SCC) [Sardashti et al. 2014] also uses super-block tags, but eliminates DCC's backward pointers. SCC makes tag-data mapping simpler by fixing a block's possible positions in the cache. Depending on the compressibility of the data, SCC will map a block to a particular set of cache ways. This limits the effective associativity, thus potentially increasing the cache miss rate. SCC compensates using a skewed-associative design, where each (group of) cache way(s) is indexed using a different hash function [Seznec 1993; Seznec 2004]. Prior results show that skewing roughly doubles the effective cache associativity. Overall, SCC achieves similar performance and energy benefits as DCC, but with lower complexity and area overhead.

However, SCC has several limitations that keep it from being an ideal compressed cache design. Skewed associativity has not found wide-spread adoption by industry. Since each cache way is indexed by a different hash function, SCC needs a separate address decoder for each tag way, increasing area and complexity. SCC also cannot use modern cache replacement policies since skewing eliminates the conventional notion of a cache set. Thus, this makes it difficult or impossible to

exploit many of the modern LLC cache replacement policies that have been proposed in the past decade [Jaleel et al. 2010].

In this paper, we propose YACC (Yet Another Compressed Cache). The main goal of YACC is to achieve the effectiveness of the DCC and SCC proposals, but with a simple and low overhead design. DCC requires changes to the tag array (to add back pointers as well as other state bits) and data array (to access sub-blocks). SCC requires changes to only the tag array, but requires separate decoders to implement the different hash functions as well as adding additional state bits. On the other hand, YACC can be used with a largely unmodified tag array and an unmodified data array.

To achieve these goals, YACC inherits the main sources of efficiencies in DCC and SCC:

- YACC uses super-block tags (i.e., each tag tracks up to 4 neighboring blocks) to lower tag area overhead.
- YACC compacts neighboring blocks with similar compression ratios in one data entry (i.e., 64 bytes) and tracks them with a super-block tag. In this way, it can track up to four times more compressed blocks with low tag area overhead.
- YACC keeps a compressed block within a data entry in the cache (i.e., not scattering sub-blocks across a set), eliminating the need for an alignment network.

On the other hand, YACC addresses the remaining sources of complexities in previous work:

- Unlike SCC and DCC, YACC's cache layout is similar to conventional caches, with a largely unmodified tag array and unmodified data array. Independent of a block's compression ratio, YACC allows it be allocated in any way of a conventional set.
- Unlike SCC and DCC, YACC's simple, conventional tag mapping allows designers to implement the whole spectrum of recently-proposed LLC replacement policies.
- Compared to DCC, YACC uses less area and a simpler access path, by eliminating the backward pointers and using a conventional data array.
- Unlike previous variable-size compressed caches that could store sub-blocks of a compressed block across different cache ways, YACC makes cache design simpler by storing a compressed block in one cache way. Thus, similar to a regular cache, on an access to a block, YACC activates only one cache way, and so does not need any extra alignment network.
- Unlike SCC, YACC eliminates the extra area and complexity of skewing.
- Unlike either DCC or SCC, YACC can also use modern replacement policies, such as RRIP [Jaleel et al. 2010], further reducing cache design complexity.
- YACC also provides some additional mechanisms to improve effective cache efficiencies. For example, YACC allows in-place expansion of a block, if the block size grows and it is the only resident of a data entry. In such a case, SCC will invalidate the block, and reallocate that to a different data entry, incurring higher overheads. In addition, unlike SCC, which stores blocks of a super-block in order, YACC could store non-adjacent blocks of a super-block together.

On our set of benchmarks, YACC improves system performance and energy by on average 8% and 6%, respectively, and up to 26% and 20%, respectively, compared to a regular uncompressed 8MB cache. Similar to DCC or SCC, YACC achieves comparable performance and energy as a conventional cache of twice the size, using far less area.

This paper is organized as follows. We discuss basics of compressed caching and related work in Section 2. Section 3 presents our proposal, YACC. Section 4 explains our simulation infrastructure and workloads. In Section 5, we present our evaluations. Finally, Section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

The past decade and a half have seen a number of compressed cache proposals. Early work [Kim et al. 2002; Lee et al. 2000] limits the maximum benefit of compression to a factor of two, by providing exactly twice as many tags and always allocating at least a half cache block regardless of the data compressibility. More recently, DCC [Sardashti and Wood 2013] proposes using decoupled super-blocks to address these limits, but at some extra cost and complexity due to the extra level of indirection (i.e., backward pointers) and the need to separately manage block and super-block replacements. SCC [Sardashti et al. 2014] eliminates DCC's backward pointers and simplifies cache replacement, but adds the complexity of skewed associativity, complicating the tag array design and limiting the choice of replacement policy. In this section, we summarize the basics of compressed cache designs, and discuss these previous works and their trade-offs in more detail. In the next section, we explain our proposed YACC design which addresses these remaining issues with SCC, achieving the same benefits from compression with a much simpler design.

2.1 Compressed Cache Fundamentals

In general for a compressed cache design, designers pick one or more compression algorithms to represent a cache block's data using fewer bits and use a compaction mechanism to store variable-size compressed blocks in the cache. Several compression algorithms have been proposed that trade-off low decompression latency (on the cache's critical access path), low complexity (area overhead), and compressibility for small cache blocks [Alameldeen and Wood 2004; Pekhimenko et al. 2012; Chen et al. 2010].

Our work is largely independent of the specific compression algorithm. In this study, we use the C-PACK+Z algorithm [Sardashti and Wood 2013] that is a variation of the C-PACK algorithm [Chen et al. 2010] with support to detect zero blocks. C-PACK+Z has been shown to have low hardware overheads and decompression latency, with fairly high compression ratio [Chen et al. 2010; Sardashti and Wood 2013]. C-PACK+Z uses a combination of spatial-value compression (e.g., representing values near zero using fewer bits) and temporal-value compression (e.g., using a 16-entry dictionary to replace (partially) recurring values with a table index). In recent CMOS technologies, CPACK+Z has a decompression latency of 9 cycles, which is low enough for use with a compressed LLC. There are other low-overhead algorithms appropriate for cache compression which we expect would give largely equivalent results.

Given a compression algorithm, a compaction mechanism is needed to fit more compressed blocks in the same space than a regular uncompressed cache. Such a mechanism needs tags to track the additional blocks and a means to map between a block's tag and its corresponding data. The second issue arises because compressed

Table 1: Compressed Cache Taxonomy

	FixedC	VSC	IIC-C	DCC	SCC	YACC
Super-Block Tags	✗	✗	✗	✓	✓	✓
Direct Tag-Data Mapping	✓	✗	✗	✗	✓	✓
Variable-Size Blocks	✗	✓	✓	✓	✓	✓
Re-compaction Free	✓	✗	✓	✓	✓	✓
Flexible Replacement Policies	✓	✓	✗	✗	✗	✓
No Extra Data Alignment	✓	✗	✗	✗	✓	✓
Single Tag Decoder	✓	✓	✓	✓	✗	✓

blocks will have different sizes, breaking the traditional direct one-to-one, tag-data mapping of conventional caches. Thus a compaction mechanism needs to address the following issues: (1) how to provide extra tags in the tag array at a reasonable storage overhead and (2) how to allocate compressed blocks and provide an efficient tag-data mapping? This work focuses on providing a simple yet effective compaction mechanism.

2.2 Design Trade-offs in Compressed Caches

Previous work has proposed a number of design alternatives to address these issues, each with different trade-offs. We summarize them in Table 1 and discuss them below.

Super-Block vs. Block Tags: In general there are two main approaches to provide extra tags: simply increasing the number of tags, or managing the tag array at super-block granularity. Several designs [Alameldeen and Wood 2004; Kim et al. 2002; Baek et al. 2013] simply increase the number of tags, for example doubling the number of tags per set. However, doubling the number of tags increases the overall cache size by 6—7% for 64-byte blocks, making this approach unattractive for designs that target more than twice as many tags. Alternatively, DCC and SCC use super-block tags to track multiple neighboring blocks (e.g., up to 4 blocks) with a single tag [Sardashti and Wood 2013]. Super-blocks can substantially reduce tag overhead but may suffer significant internal fragmentation for workloads that lack spatial locality.

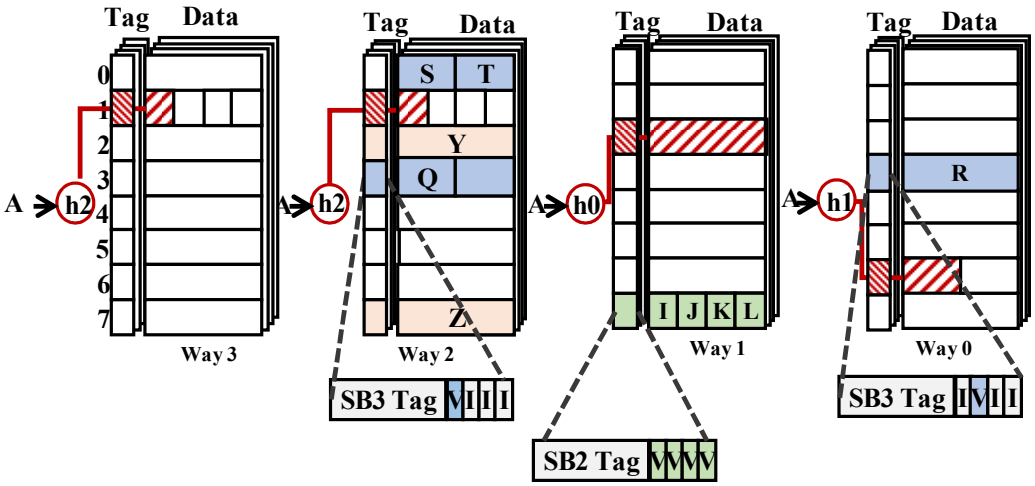
Direct vs. Decoupled Tag-Data Mapping: Given a matched tag, the next step is to locate the corresponding block in the data array. We can categorize existing techniques into: (conventional) direct tag-data mapping and decoupled tag-data mapping. Direct mapping associates a tag with a particular data entry, similar to regular uncompressed caches. Conversely, decoupled mapping techniques add a level

of indirection, and so require extra metadata to locate a block in the data array. This metadata can either be an explicit pointer (either forward or backward) or encode other information such as the allocation size that can be used to compute an offset. The level of indirection in the decoupled mapping may allow multiple blocks to be stored more compactly in the data array, but comes with additional area overhead for the metadata and design complexity. Conversely, using direct tag-data mapping makes compressed cache designs simpler, as the matching tag's location uniquely identifies the location of the block in the data array.

Variable-Size vs. Fixed-Sized Compressed Blocks: Early design proposals supported only a single, fixed-size compressed block (i.e., half of an uncompressed block) even if data were highly compressible. These proposals, which we call FixedC [Kim et al. 2002; Lee et al. 2000]), facilitate direct tag-data mappings, but limit the benefit of compression to at most a factor of two and often achieve much less. Conversely, the effective capacity of the cache can be increased significantly using variable-size allocation for compressed blocks. Typically, this involves allocating a variable number of fixed-size sub-blocks (e.g., 8—16 bytes each) to hold a block's data. In some designs, sub-blocks must be contiguously allocated (e.g., VSC [Alameldeen and Wood 2004]) and in others they may be non-contiguous (e.g., IIC-C [Hallnor and Reinhardt 2005] and DCC [Sardashti and Wood 2013]). Variable-size allocation reduces internal fragmentation and can substantially increase the effective cache capacity.

Re-Compaction Overhead: Updating a compressed cache block may change the compressibility of its data and thus the amount of space needed to store the block. Hence an update to a compressed cache may result in the need to write back one or more cache blocks to make room for a new, larger (and potentially uncompressed block). This problem arises with all compressed caches, but occurs more frequently with variable-size compressed blocks due to their lower internal fragmentation. It is particularly significant with VSC, which requires that all allocated sub-blocks be contiguous; thus changing the size of one block may require moving all the allocated sub-blocks in a cache set. IIC-C and DCC use forward and backward pointers, respectively, to eliminate unnecessary re-compaction overheads. As updates happen frequently in some workloads, unnecessary re-compaction can add significant overheads to cache dynamic energy.

Data Alignment: Cache compression complicates the data array access since some blocks will be uncompressed and others require fetching, and potentially aligning, a variable number of bits, bytes, or sub-blocks. The greater the flexibility in data allocation, the greater the complexity and delay of the resulting alignment network. Most prior work limits the allocation granularity to sub-blocks, ranging in size from 8-bytes to 32-bytes, but differ in how they allocate sub-blocks across a set. VSC and DCC treat the entire cache set as a pool of sub-blocks, eliminating the conventional notion of a (data) cache way. While VSC requires a complex data alignment network, Sardashti and Wood explained how to extend the AMD Bulldozer cache design to read out sub-blocks without an alignment network [Sardashti and Wood 2013]. However, their technique still requires changes to the data array and only works when the LLC datapath is no larger than the sub-block size. Conversely, SCC and YACC support variable-size compression, but always access a full cache block and thus require no changes to the data array or explicit data alignment network.



SB1: A_ _ _; SB2: IJKL; SB3: QRST; SB4: YZ_ _ _

Fig. 1. Skewed Compressed Cache (SCC)

2.3 SCC: A State-of-the-art Compressed Cache

SCC [Sardashti et al. 2014] is a state-of-the-art compressed cache design that picks the best of the alternatives discussed in Section 2.2. SCC uses super-block tags and variable-size compressed blocks to allow many compressed blocks to be packed in the cache, while incurring only low tag overhead. Furthermore, SCC uses direct tag-data mapping to eliminate the need for explicit pointers, substantially reducing the amount of additional metadata. And, finally, SCC eliminates unnecessary re-compaction on updates, never changing a cache block's storage location because a *different* block changes size.

SCC does this using a novel sparse super-block tag, which tracks anywhere from one block to all blocks in a super-block, depending upon their compressibility. As illustrated in Figure 1, a single sparse super-block tag can track: all four blocks in a super-block, if each is compressible to 16 bytes (e.g., blocks I, J, K, and L); two adjacent blocks, if each is compressible to 32 bytes (e.g., blocks S and T); and only one block, if it is not compressible (e.g., block Y). By allowing variable-size compressed blocks—16, 32, and (uncompressed) 64 bytes—SCC is able to tightly compact blocks and achieve high compression effectiveness.

Figure 2 (a) shows one set of an SCC tag array and its corresponding data set for a 4-

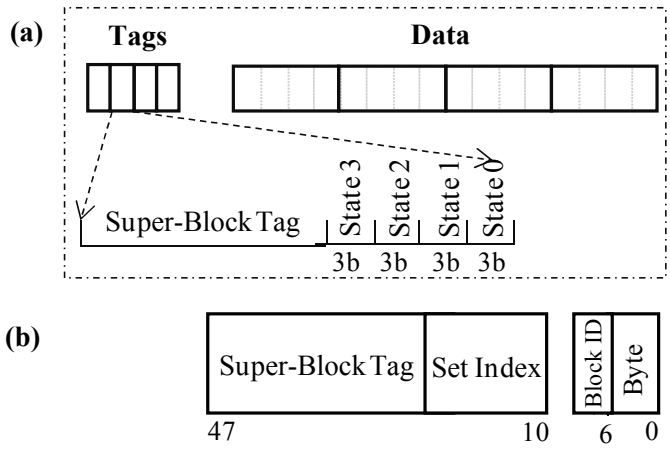


Fig. 2. (a) One set of SCC (b) SCC Address Partitioning

way associative cache. Like a regular cache, SCC has the same number of tags as data entries (e.g., 4 tags and 4 data entries per set). However, as shown in Figure 2 (a), each tag entry includes the super-block tag address and per-block coherency/valid states. For this example, compared to a regular cache, each super-block tag entry only requires an additional 7 bits of storage: 9 additional coherency/valid bits for the three additional blocks, but two fewer tag address bits, since each tag maps a 256-byte (4 block) super-block.

Figure 2 (b) shows the partitioning of addresses in SCC. Assuming a 64-byte block, the first 5 bits of address is used to access a particular byte in a block. The next two bits (Block ID) is then used to distinguish a block in a 4-block super-block. As with a regular cache, higher order bits will be used to index a set, and as tag address. As we explain later, for this mapping, SCC will use A_9A_8 for mapping a block to some ways. Unlike a regular cache which can allocate a block in any cache way, SCC must use a block's compressibility to determine where to allocate it.

$$W_1W_0 = A_9A_8 \text{ ^ } CF_1CF_0 \qquad (1)$$

Equation (1) shows that the block compression factor (CF_1CF_0) is exclusive-ORed with two address bits (A_9A_8) to select the appropriate way group (W_1W_0). The block compression factor is zero if the block is not compressible (CF_1CF_0 = 0b00), one if compressible to 32 bytes (CF_1CF_0 = 0b01), and two or three if compressible to 16 bytes (CF_1CF_0 = 0b1X)[1]. Blocks compressible to 16 bytes will map to two way groups since CF_0, and thus W_0, are "don't cares". Because SCC uses address bits A_9A_8 to select the way group, it does not use them as part of the set index. This also ensures that even if all cache blocks are uncompressible (CF == 0), they will spread out among all cache ways.

Figure 1 illustrates the interaction of skewing with compression. Block A maps to way group 0 (4 ways), way group 1 (4 ways), or way groups 2 and 3 (8 ways), when it compresses to 32 bytes, 64 bytes (uncompressed), and 16 bytes, respectively.

$$\text{Set Index} = \begin{cases} h_0(\{A_{49}\text{—}A_{10}, A_7A_6\}) & \text{if CF==0} \\ h_1(\{A_{49}\text{—}A_{10}, A_7\}) & \text{if CF==1} \\ h_2(\{A_{49}\text{—}A_{10}\}) & \text{if CF==2 or 3} \end{cases} \qquad (2)$$

SCC uses different set index functions to prevent conflicts between blocks in the same super-block. Just using bit selection, e.g., the consecutive bits beginning with A_{10}, would result in all blocks in the same super-block mapping to the same set in a way group, resulting in unnecessary conflicts. For example, if none of the blocks were compressible, then all uncompressed blocks would compete for the entries in the selected way group (in Figure 2). To prevent this, SCC uses the index hash functions shown in Equation (2), which draw address bits from the Block ID for the less compressible blocks. These functions map neighboring blocks to the same set only if they can share a data entry (based on their compression factor). SCC also uses different hash functions [Seznec 2004] for different ways in the same way group, to further reduce the possibility of conflicts.

[1] The original SCC paper used a compression factor of two for blocks compressible to 16 bytes and three for blocks compressed to 8 bytes. However, simulation results generally show it is better to compress to a minimum size of 16 bytes.

Within a 64-byte data entry, a compressed block's location depends only on its compression factor and address, eliminating the need for extra metadata such as forward or backwards pointers. Equation 3 shows the function to compute the byte offset for a compressed block within a data entry.

$$\text{Byte Offset} = \begin{cases} \text{none} & \text{if CF==0} \\ A_6 << 5 & \text{if CF==1} \\ A_7A_6 << 4 & \text{if CF==2 or 3} \end{cases} \quad (3)$$

On a cache lookup, SCC must check all the block's corresponding positions in all cache ways since the block's compressibility is not known. To determine which index hash function to use for each way, SCC uses Equation (4), the inverse of Equation (1). For example, in Figure 2, when accessing block *A*, the tag entries in set #1 of way groups #2 and #3, set #2 of way group #1, and set #6 of way group #0 (i.e., all hatched red tag entries) are checked for a possible match. A cache hit occurs if its encompassing super-block is present (i.e., a sparse super-block tag match), and the block state is valid. On a read hit, SCC uses the compression factor and appropriate address bits (using Equation (3)) to determine which of the corresponding sub-blocks should be read from the data array.

$$CF_1CF_0 = A_9A_8 \wedge W_1W_0 \quad (4)$$

SCC achieves performance comparable to the more complex DCC algorithm, but does so with significantly less metadata overhead due to the elimination of the backward pointer metadata. SCC also simplifies the replacement policy, compared to DCC, as it always replaces an entire sparse super-block. Like DCC, SCC must decompress compressed blocks before writing them back to memory and uses a per-bank writeback buffer to reduce contention for the bank's decompression hardware. Because SCC has multiple banks, and thus multiple decompressors, we observe little contention in practice.

2.4 Limitations of SCC

Despite being an improvement over DCC, SCC has several limitations that keep it from being an ideal compressed cache design. We discuss these in turn below.

Limited Effective Associativity: SCC limits the effective associativity of the cache. This occurs because SCC uses a block's compression factor to determine in which subset of ways it should be allocated. Thus in a 16-way SCC, a particular block may only be allocated to one of 4 ways (or 8 ways, if compressible to 16 bytes), substantially reducing the effective associativity of the cache. Additional skewing (within a way group) can help reduce conflict misses, but adds additional complexity and overheads. Furthermore, additional intra-way group skewing does not provide uniform benefit for all workloads.

Multiple Tag Decoders: Address lookups to a regular, non-skewed cache require only a single address decoder to select the appropriate set in the tag array. Thus all tags can be checked in parallel with one decode operation. Skewed-associative caches need a separate decoder for each hash function. Thus SCC requires a separate decoder per way group (e.g., one per compression factor). To address intra-way group skewing (in addition to the inter-way groups skewing based

on compression factor), SCC would need one decoder per cache way. Increasing the number of decoders increases both the area, static and dynamic energy of the tag array.

Constrained Replacement Policies: Ideally a compressed cache should be able to use any replacement policy; however, that is not the case for many proposals. DCC must make separate, but dependent decisions on when to replace blocks and super-blocks. For example, allocating space for a single cache block could result in DCC replacing multiple blocks and multiple super-blocks. SCC simplifies replacement compared to DCC by always replacing an entire super-block. However, it inherits all the replacement issues associated with skewed associative caches. Skewing limits the choice of replacement policy because each cache way (or way group) uses a different index, eliminating the traditional notion of a cache set. Thus, skewed-associative caches (including SCC) are not directly compatible with set-based replacement policies, such as conventional pseudo-LRU as well as modern scan-resistant replacement policies (e.g., RRIP). Skewed-associative caches can use the replacement policy from Zcache [Sanchez and Kozyrakis 2010], but at the expense of significant complexity.

Difficult to Understand Behavior: Skewed-associative caches tend to eliminate conflict misses on average [Seznec 1993], but the multiple hash functions make it difficult to guarantee that two addresses will not conflict. Thus a compiler or sophisticated programmer may find it difficult to explicitly manage cache capacity, as is often done when tiling scientific workloads.

Despite the potential for skewed-associative caches to reduce conflict misses, they have not—to our knowledge—been adopted in commercial last-level caches. We believe this reluctance stems, at least in part, from a combination of the last three limitations discussed above. And, to the extent that is reluctance continues, we find this compelling motivation to develop a non-skewed version of SCC.

3. YACC : YET ANOTHER COMPRESSED CACHE

YACC is a new compressed cache design that seeks to preserve the good aspects of SCC, while eliminating the limitations associated with skewing. YACC essentially de-skews SCC, increasing the effective associativity to that of a conventional cache. In addition to simplifying the tag array, YACC allows the use of any replacement policy and (by eliminating skewing) leads to more predictable behavior.

3.1 YACC Architecture

Figure 3 (a) shows a high-level overview of an 8-way associative YACC design. Like SCC, YACC uses sparse super-block tags to exploit both spatial locality (i.e., there are many neighboring blocks in the cache) and compression locality (i.e., neighboring blocks tend to have similar compressibility). YACC stores neighboring blocks in one data entry if they have similar compressibility and could fit in one data entry.

Figure 3 (b) shows how YACC partitions address. For the sake of simplicity, throughout this paper the (uncompressed) data block size is 64 bytes and super-blocks have 4 blocks. Thus the lowest 6 bits in the address (bits A_5–A_0, labeled Byte in Figure 3(b)) refer to the byte offset in a block. The next two address bits (bits A_7–A_6, labeled Blk ID in Figure 3(b)) identify which block is accessed in a super-block.

Unlike SCC's skewed-associative design, YACC keeps the tag-data mapping simple by using a simple bit-selection hash function to select a conventional set. Figure 3(b) illustrates that YACC uses the bits just above the Blk ID to index both the tag and data arrays. Cache hits are detected by comparing the super-block tag and checking the corresponding block's three-bit coherence/valid state.

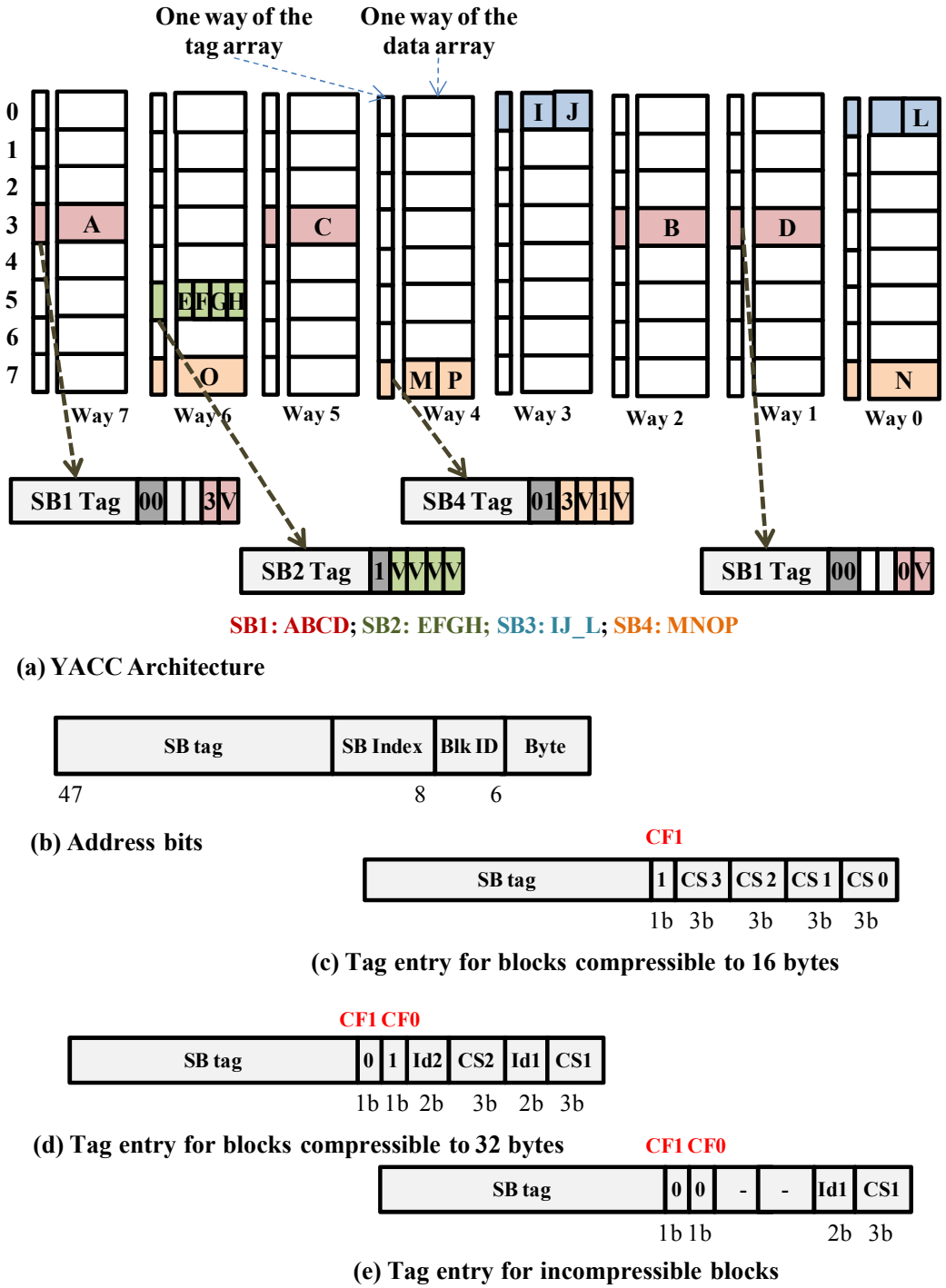


Fig. 3. YACC Architecture: in an 8-way associative cache, YACC associates one tag entry (sparse super-block tag) per data entry (i.e., 64 bytes). Each tag entry tracks up to 4 neighboring blocks, if they have similar compressibility and could fit in one 64-byte data entry (e.g., SB2). If neighboring blocks have different levels of compressibility, they would be allocated in different data entries in the same set (e.g., in SB4, M and P are stored together, while N and O are allocated separately). For each data entry, the corresponding tag indicates which blocks are present.

YACC eliminates the implementation issues that make DCC and SCC less than ideal designs. YACC uses a conventional super-block tag array and an unmodified data array. In contrast, DCC requires significant changes to both the tag array (to add back pointers as well as other state bits) and data array (to access sub-blocks). SCC requires changes to only the tag array, but requires separate decoders to implement the different hash functions as well as adding additional state bits. YACC only requires the addition of a few state bits to the tag array, which is a relatively simple change. We will next discuss YACC design in more detail.

3.1.1 YACC Tag Format

Like SCC, YACC uses sparse super-block tags [Sardashti et al. 2014] to map the blocks that are compressed in the corresponding data. In Figure 3(a), blocks E, F, G, and H of super-block SB2 are each compressible to one 16-byte sub-block and YACC stores all four in one 64-byte data entry. Thus this implementation of YACC has the potential to increase the cache capacity by a factor of four. Conversely, blocks M and P of super-block SB4 are each compressible only to 32-bytes and blocks O and N are not compressible; thus SB4 requires three separate data entries to hold its blocks, one for blocks M and P and one each for blocks O and N. Super-block SB1 represents the worst case, where each of its blocks A, B, C, and D require a separate data entry. Note that since all the blocks in a super-block map to the same set, workloads that exhibit spatial locality but are not compressible will have less effective associativity.

Figure 3(c-e) illustrate the structure of YACC tag entries. Each sparse super-block tag includes metadata which tightly encodes which blocks of a super-block are stored in the corresponding data entry. Each tag includes the super-block tag address, the compression factor of the blocks, and per-block coherency/valid states. To represent this information with minimum bits, YACC uses a different format for each compression factor and even exploits the "don't care" in the compression factor encoding for blocks compressible to 16-bytes. Specifically, Figure 3(c) shows that only CF1 is stored in this important case; this is possible since CF0 is a "don't care". The remaining 12 bits encode a 3-bit coherence state for each of the four sub-blocks. For example, Figure 3(a) shows that the tag for super-block SB2 has all four block states set to "V", indicating that each of blocks E, F, G, and H are valid and stored in the same data entry. If CF1 is not 1, then the next (less significant) bit is CF0 which differentiates between the uncompressed and 32-byte compressed formats. Figure 3(d) illustrates the case that CF1=0 and CF0=1 and thus the tag entry can map two data blocks compressed to 32-bytes each. Note that in addition to the two three-bit coherence fields, this format also has two two-bit index fields that identify which blocks are compressed. This allows YACC to store non-adjacent blocks from the same super-block in a single entry, a significant advantage compared to SCC which requires that they be adjacent. This is illustrated in Figure 3(a) for blocks M and P of super-block SB2; block M has index 3 and block P has index 0. Finally, Figure 3(e) illustrates the case that the block is uncompressible, with CF1=0 and CF0=0. In this case there is a single coherence state field and a single block index. Figure 3(a) shows that the tags for blocks A or D use this format.

3.1.2 How to allocate compressed blocks?

Like SCC, YACC uses a direct tag-data mapping to keep compressed block allocation simple. It compresses blocks into a power of two numbers of sub-blocks (e.g., one, two, or four 16-byte sub-blocks). It then stores blocks with the same compression factor in a single data entry if they fit (e.g., blocks E,F,G,H of SB2). Otherwise, YACC stores them in one or more other data entries in the same data set. For example, blocks I, J,

and L from SB3 are each compressible to half a block. YACC fits I and J in one data entry (way 3 of set 0), and stores L in a separate data entry in the same set (way 0 of set 0). Later if we access block K from SB3, YACC could compress and store it in the same data entry as L if it is compressible to 32-bytes.

In case that blocks belonging to the same superblock are not compressible, such as blocks A, B, C, D of SB1, YACC stores these blocks separately in different data entries (e.g., in ways 7, 5, 2, and 1) in the same cache set (set #3). In this way, these blocks compete for the same set in the cache. Thus, an application featuring low data compressibility and high spatial locality, might suffer from a limited visible associativity, as we will discuss in the evaluation section.

No cache skewing: A major goal of YACC is to eliminate the use of skewed associativity, which complicates the design of SCC. In SCC, a block is mapped to particular way group depending on its address and compressibility. Because this limits the effective associativity (e.g., an uncompressed block can only map to one way group) additional skewing is done within a way group.

Figure 4 presents a high-level comparison of SCC and YACC. On a lookup, where the block size is unknown, SCC checks all possible positions of the block (8 in the illustrated 8-way associative cache). SCC skews each cache way with a different hash function. For example in Figure 4 (b), the colored blocks (yellow and red) show the tags being checked on a lookup. Note that because of skewing each tag is in a different row. Therefore, SCC requires eight decoders on the tag array, one per cache way. This complicates the tag array design, area, and possibly layout and routing. On the other hand, YACC stores a block in any cache way in a given cache set (indexed by super-block index bits). For example, in Figure 4 (a), on a lookup, all tags of set 1 (in yellow and red) will be checked for a possible hit. In this way, YACC uses a conventional tag array design, only requiring one decoder for the whole tag array. Note that the extra decoders in SCC are only needed for the tag array and not the data array. In both SCC and YACC, when a tag matches, we will then index and access the corresponding data entry in the data array (colored in red in Figure 4).

In addition to complicating tag array design, skewing would also limit the choice of replacement policy. In SCC, any block could map to a different set of rows. Thus, there is no fixed notion of a cache set, making it difficult to employ many replacement policies. On the other hand, YACC can use any replacement policy, such as RRIP [Jaleel et al. 2010].

In-place block expansion: YACC allows in-place expansion of a block on a write-back from lower cache levels, if the block size grows and it is the only resident of a data entry. For example, if block L's size grows, so that it becomes uncompressible (requiring 64 bytes), YACC will store it in the same data entry. It only changes its status in the corresponding tag. SCC will need to invalidate, and reallocate that block to a different data entry on a block update.

Storing non-adjacent blocks together: In order to compact more blocks, YACC does not necessarily compact blocks of a super-block in the same order. For example, blocks M and P from SB4 are compressible to 32 bytes each. Although they are not contiguous neighbors, YACC would still pack them together. Previous work, SCC, does not support this mode as it stores blocks in strict order. In a similar situation, SCC would map these blocks to different entry, and would allocate a data entry for each.

For example, SCC would allocate M and P in different data entries, using 128 bytes (2*64 bytes) instead of 64 bytes (2*32 bytes) in YACC. In order to locate these blocks, YACC encodes the tag entry differently for difference compression factors

(shown in Figure 3). The tag entry includes block id (2 bits) and coherence state (3 bits) of the blocks stored into the lower or the upper half of the corresponding data entry. For example, the tag entry in way #4 of set #7 indicates that block #3 of SB4 (block M) and block #1 of SB4 (block P) are stored in upper and lower halves of the corresponding data entry, respectively.

3.2 YACC Cache Operations

3.2.1 Cache Read

Figure 5 shows a block diagram of main cache operations in YACC. On a cache read, YACC indexes a set of tag array using super-block index bits from the address. For example, in Figure 3, to read block A of SB1, YACC will index set #3. It then checks all tag entries in that set for a possible hit. A cache hit occurs if the tag address matches, and the corresponding coherence state is valid. For example, YACC finds block A in way #7 of set #3, as the super-block tag address in that tag entry matches

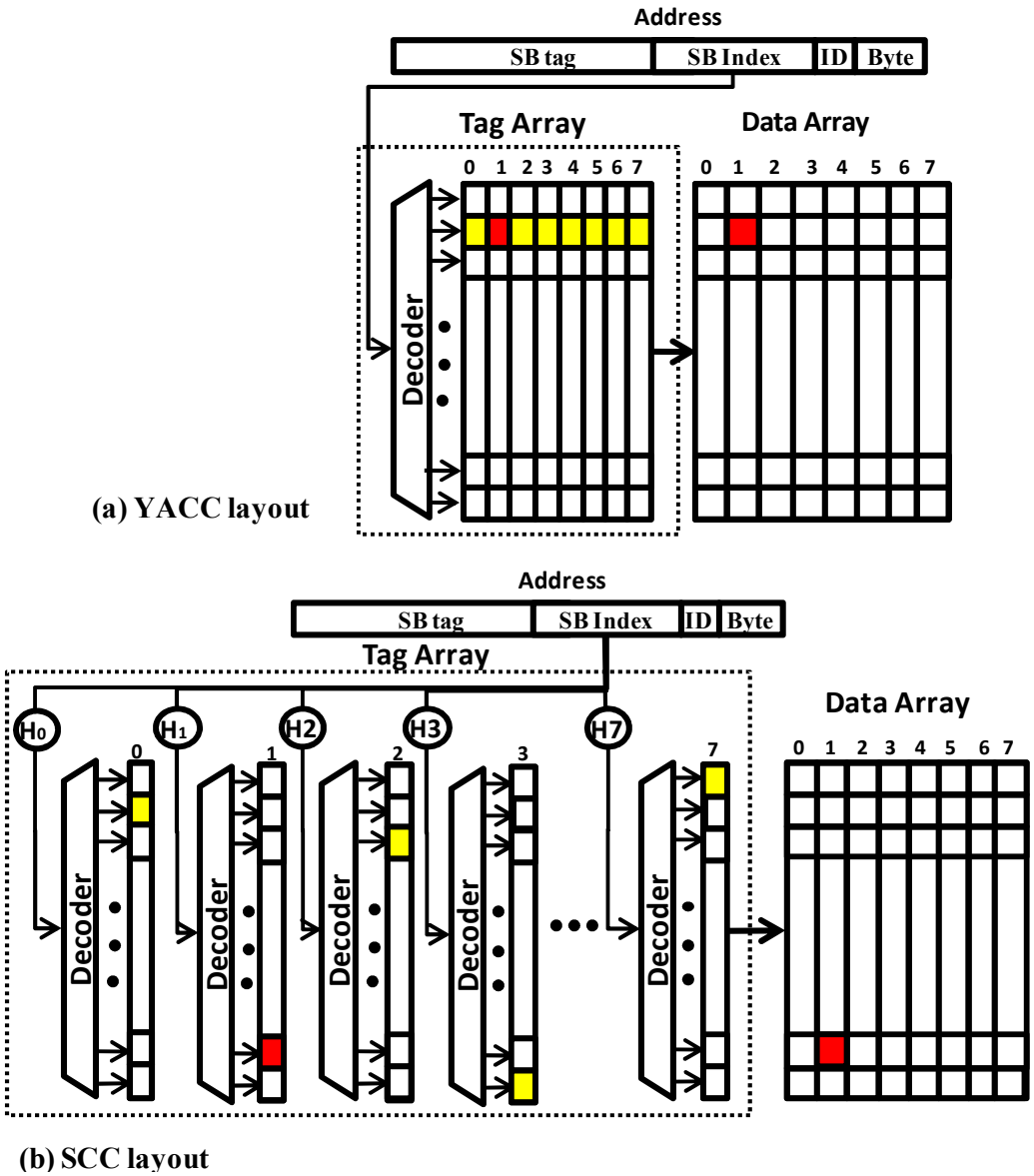


Fig. 4. Cache physical layout with YACC (a), and SCC (b)

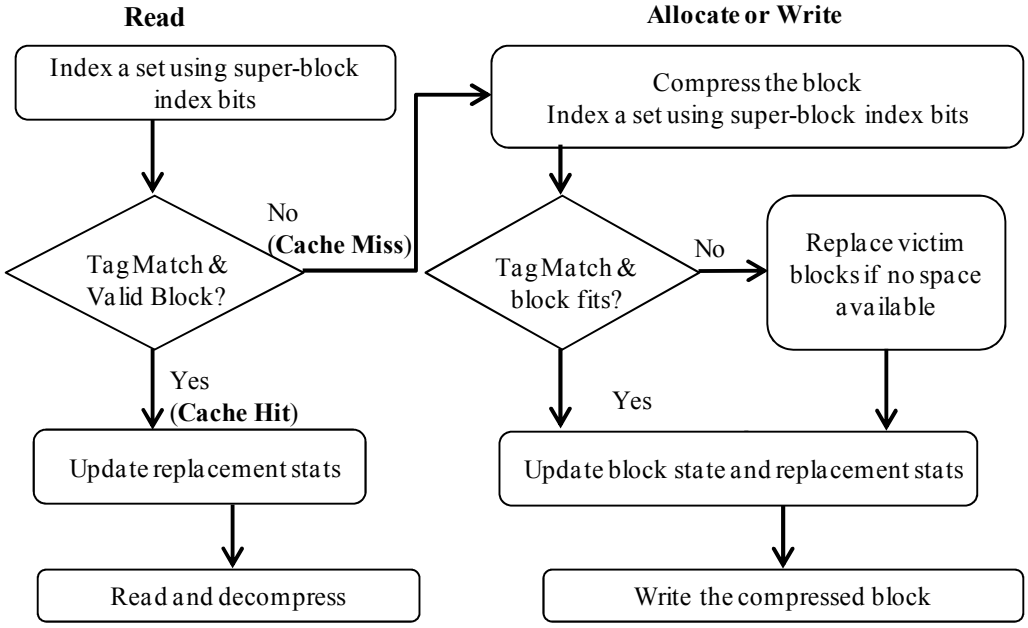


Fig. 5. YACC cache operations.

with SB1 tag address, and the tag entry indicate the block A is valid (block #3 exists in Valid state).

Note that since YACC maps all blocks of a super-block to the same set, it is possible to have more than one tag entry matching the super-block tag address, but only a single corresponding valid tag match. For example, the tag entry in way #1 of set #3 also tracks part of SB1 (i.e., block D), so it also has the same super-block tag address but block A is invalid. On a read hit, YACC would read out and decompress the corresponding sub-blocks from the data array. On a cache miss, YACC would allocate the block in the cache.

3.2.2 Cache Allocate and Cache Replacement

On a cache allocate (e.g., caused by a cache miss), a cache write or write-back, YACC first compresses the block. It then indexes a set of the tag array using the super-block index bits from the address. In that set, YACC first tries to fit the block in an already existing SB. To do so, YACC first checks for a tag entry with the same super-block tag address and compression factor. If so, it then checks to see if the block can fit in the corresponding data entry. For example, YACC can allocate block K (from SB3) in the same entry as block L, if K is also compressible to half.

On a write-back (or update) to an existing block, if the block size grows, YACC might need to invalidate the previous version of this block before re-allocating it. If the block is the only block in that entry, such as block L, YACC will not invalidate or re-allocate it. It simply stores the block in the same entry, and only updates the tag. Otherwise, if it does not fit in its previous entry, it would invalidate and allocate it in the cache as just explained.

In case there is no matching tag entry with enough space for the accessing block, YACC needs to replace a victim sparse super-block first before allocating this block. Finding the victim tag is straightforward, and basically similar to a regular cache. Depending on the replacement policy, it finds the victim tag, and evicts the blocks

resident in its entry. For example, if YACC picks way #4 of set #7 as victim, it would evicts blocks M and P of SB4, and free that entry. Note that other blocks of SB4 (i.e., blocks N, and O), which are not resident of this particular entry, will still stay in the cache.

When replacing a victim super-block, similar to a regular cache, YACC writes back dirty blocks to the main memory. Since we are assuming blocks are stored in uncompressed format in the memory, YACC needs to decompress compressed blocks when evicting them. Similar to DCC and SCC, YACC uses one compression unit and one decompression unit per cache bank. Thus, while decompressing and evicting blocks, other cache banks could still be used. In addition, when evicting a super-block, we can read and copy the whole data block (64 bytes) once to a local buffer, then decompress and send its blocks to memory in the background.

4. METHODOLOGY

To evaluate YACC, we use the same evaluation framework that was used to evaluate SCC and DCC. We use full-system cycle-accurate GEMS simulator [Martin et al. 2005]. We model YACC with an 8-core multicore system with OOO cores, per-core private L1 and L2 caches, and one shared last level cache (L3). We implement YACC and other compressed caches at the L3. Table 2 shows the main parameters. We use 64-byte cache block sizes. For YACC, SCC [Sardashti et al. 2014], and DCC [Sardashti and Wood 2013], we use 4-block super-blocks (each tag tracks 1-4 neighbors), and 16-byte sub-blocks (i.e., each block compress to 0-4 sub-blocks).

We use CACTI 6.5 [CACTI] to model power at 32nm. We also use a detailed DRAM power model developed based on the Micron Corporation power model [Micron 2007] with energy per operation listed in Table 2. In this section, we report total system energy that include energy consumption of processors (cores + caches), on chip network (using Orion), and off chip memory.

4.1 Applications

We use several applications with different characteristics from SPEC OMP [Aslot et al. 2001], PARSEC [Bienia and Li 2009], commercial workloads [Alameldeen et al. 2005], and SPEC CPU 2006. From SPEC CPU 2006 benchmarks, we run mixes (mix1 – mix8) of multi-programmed workloads from memory-bound and compute-bound. For example, for omnetpp-lbm, we run four copies of each benchmark. Table 3 shows

Table 2. Simulation Parameters

Processors	8, 3.2 GHz, 4-wide issue, out-of-order
L1 Caches	32 KB 8-way split, 2 cycles
L2 Caches	256 KB 8-way, 10 cycles
L3 Cache	8 MB 16-way, 8 banks, 27 cycles
Memory	4GB, 16 Banks, 800 MHz DDR3. 60.35nJ per Read, 66.5nJ per Write, and 4.25W static power.
Block Size	64 bytes
Super-Block Size	4-block super-blocks
Sub-block Size	16 bytes

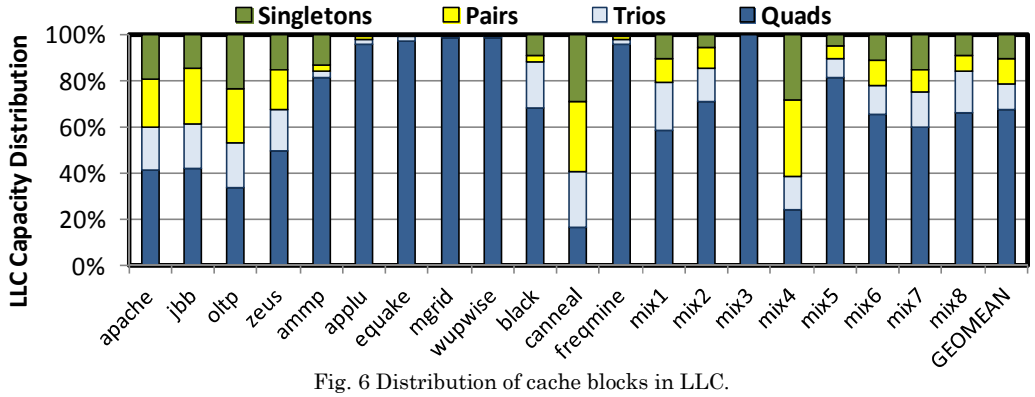


Fig. 6 Distribution of cache blocks in LLC.

our applications. We classify these workloads into: low memory intensive, medium memory intensive, and high memory intensive based on their LLC MPKI (Misses per Kilo executed Instructions) for the Baseline configuration (a regular uncompressed LLC). We classify a workload as low memory intensive If LLC MPKI is lower than one, as medium memory intensive if LLC MPKI is between one and five, and as high memory intensive if MPKI is over five.

Figure 6 shows the distribution of neighboring blocks in a conventional LLC with a tag per 64-byte block. Neighboring blocks are defined as those in a 4-block aligned super-block (i.e., aligned 256-byte region). The graph shows the fraction of blocks that are part of a Quad (all four blocks in a super-block co-reside in the cache), Trios

Table 3. Applications

Application	**Compression Ratio**	**Baseline LLC MPKI**	**Category**
ammp	3.0	0.01	Low Mem Intensive
Blackscholes	4.0	0.13	
canneal	2.8	0.51	
freqmine	3.3	0.65	
bzip2 (mix1)	4.0	1.7	Medium Mem Intensive
equake	5.0	2.2	
oltp	2.0	2.3	
jbb	2.6	2.7	
wupwise	1.3	4.3	
gcc-omnetpp-mcf-bwaves-lbm-milc-cactus-bzip (mix7)	3.9	8.4	High Mem Intensive
libquantum-bzip2 (mix2)	3.7	9.3	
astar-bwaves (mix5)	3.8	9.3	
zeus	2.9	9.3	
gcc-166 (mix4)	4.2	10.1	
apache	2.8	10.6	
omnetpp-lbm(mix8)	4.7	11.2	
cactus-mcf-milc-bwaves (mix6)	4.8	13.4	
applu	1.7	25.9	
libquantum(mix3)	4.0	43.9	

(three blocks out of four co-reside), Pairs (two blocks out of four co-reside), and Singletons (only one block out of four resides in the cache). Pairs and Trios are not necessarily contiguous blocks, but represent two or three blocks, respectively, that could share a super-block tag. Although access patterns differ, the majority of cache blocks reside as part of a Quad, Trio, or Pair. For applications with streaming access patterns (e.g. mgrid) Quads account for essentially all the blocks. Other workloads exhibit up to 29% singletons (canneal), but Quads or Trios account for over 50% of blocks for all but two of our workloads (canneal and gcc).

We run each workload for over 500M instructions (1 billion for several workloads). We use warmed up caches, and fast forward for about 100M instructions. To address workload variability, we simulate each workload for a fixed number of work units (e.g., transactions) and report the average over multiple runs [Alameldeen and Wood 2003]. Table 3 also shows the compression ratio (original size/compressed size) for each workload using C-PACK+Z algorithm [Sardashti and Wood 2013; Chen et al. 2010].

4.2 Configurations

We evaluate the following configurations for LLC:

- **Baseline** is a conventional uncompressed 16-way 8MB LLC.
- **2X Baseline** is a conventional 32-way 16MB LLC.
- **DCC** models Decoupled Compressed Cache [Sardashti and Wood 2013] with 4-block super-blocks, and 16-byte sub-blocks. We use a LRU-based replacement algorithm for both super-block and block level replacements. In addition to latencies of L3 cache shown in Table 2, DCC adds one extra cycle to cache access latency for its sub-block selection logic [Sardashti and Wood 2013]. We consider this extra overhead in our simulation.
- **SCC** models Skewed Compressed Cache [Sardashti et al. 2014] with 4-block super-blocks, and 16-byte sub-blocks. We use a LRU-based replacement algorithm for super-block replacements.
- **YACC** models our proposal with 4-block super-blocks, and 16-byte sub-blocks. We also use a LRU-based replacement algorithm for super-block replacements. Unlike DCC, SCC and YACC do not add additional latency to cache accesses as they use a direct tag-data mapping.
- **Baseline RRIP** is also an uncompressed 8-way 8MB LLC, but we use RRIP for replacement policy [Jaleel et al. 2010].
- **YACC RRIP** is similar to YACC (or YACC LRU) configuration, but we use RRIP for replacement policy [Jaleel et al. 2010].

5. EVALUATION

5.1 Hardware Overheads

Table 4 shows the area overhead of YACC and state of the art DCC and SCC. We assume a 48-bit physical address space. Compressed caches use the same data array space, but usually increase tag array space to track more blocks. In Table 4, we show the number of bits needed per set in a 16-way cache. We categorize tag space into bits needed to represent tags (e.g., tag addresses and LRU information), extra metadata needed for coherence information, and extra metadata for compression (e.g., compression factor).

Table 4. Compressed Caches Area Overhead

	Baseline	DCC	SCC	YACC
Tags per Set (bits)	16x29+4 = **468**	16x27+4+6 = (**468**-26)	16x27+4 = (**468**-32)	16x27+4 = (**468**-32)
Coherence Metadata per Set (bits)	16x3 = **48**	16x(4x3+1) = (**48**+160)	16x4x3 = (**48**+144)	16x(4x3+1) = (**48**+160)
Compression Metadata per Set (bits)	**0**	16x4x6+16x4x1 = (**0** + 448)	**0**	**0**
Total LLC Tag Array Overhead (%)	**0**	**113%**	**25%**	**28%**
Total LLC Overhead (%)	**0**	**6.7%**	**1.5%**	**1.6%**

State of the art DCC uses super-block tags to track more blocks at lower tag overhead. It uses the same number of tags, but each tracks up to 4 blocks of a super-block. The tags use fewer bits for the matching address (27-bit super-block tags versus 29-bit regular tags). DCC keeps LRU information separately per super-blocks (4 bits to find the least recently used super-block) and blocks (6 bits to find the least recently used block). Since DCC can fit up to 4 times more blocks in the same space, it keeps 4 times more coherence state (3 bits assuming MOESI). It also keeps one valid bit per super block. DCC decouples tag-data mapping, requiring extra metadata to hold the backward pointers that identify a block's location. DCC keeps one 6-bit backward pointer entry (BPE) per sub-block in a set. In addition, it stores one bit in the tag per block showing if the block is compressible or not. Overall, DCC more than doubles tag area, incurring about 6.7% area overhead on total LLC area (tag array and data array).

SCC cuts down on these extra tag bits. SCC keeps LRU information for super-blocks only, and completely eliminates extra compression metadata. In this way, SCC increases tag array area by about 24%, and total LLC area by only 1.5%. Here we are only counting the overheads in term of extra bits stored. SCC, however, requires 16 tag decoders instead of one, which we are not counting here.

In terms of tag and metadata bits stored per set, YACC is very similar to SCC. The only difference is that it keeps one extra bit per super-block, representing if the blocks are compressible to a factor of 4 (CF=4?). Compared to a conventional cache, YACC uses only 8 extra bits (10 extra coherence bits – 2 fewer tag address bits) per tag entry, as also shown in Table 4. Unlike SCC, YACC does not change tag array layout, it requires only one tag decoder and no hash function hardware to address the cache.

In terms of tag and metadata bits stored per set, YACC is very similar to SCC. The only difference is that it keeps one extra bit per super-block, representing if the blocks are compressible to a factor of 4 (CF=4?). Compared to a conventional cache, YACC uses only 8 extra bits (10 extra coherence bits – 2 fewer tag address bits) per tag entry, as also shown in Table 4. Unlike SCC, YACC does not change tag array layout, it requires only one tag decoder and no hash function hardware to address the cache

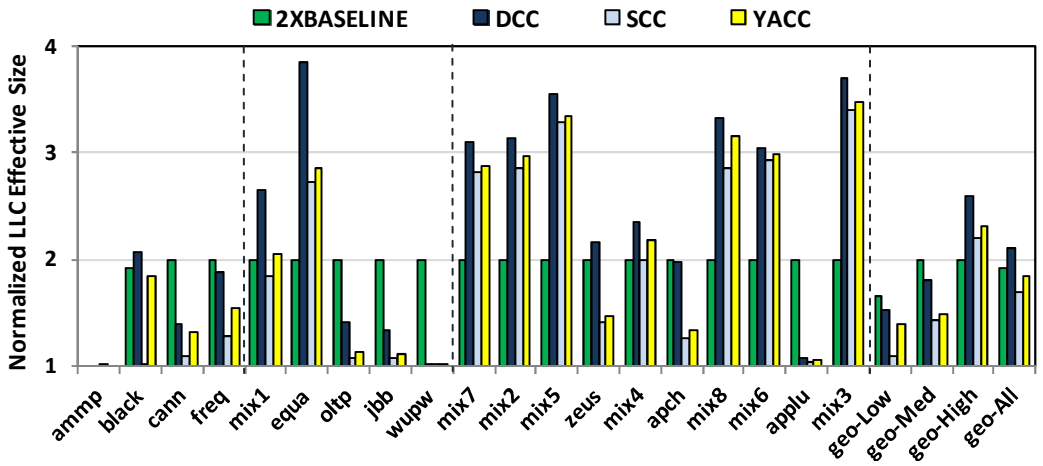


Fig. 7. Normalized LLC effective size.

5.2 Cache Effective Capacity

In general compressed caches increase cache utilization by fitting more blocks in compressed format in the same space as an uncompressed cache. Figure 7 shows the effective capacity of different designs normalized to Baseline. To calculate the effective capacity of a cache, we count the number of valid blocks in the cache when allocating a new block, and report the average number over all counts. In Figure 7, we report the effective capacity of each configuration, normalized to the Baseline. For an ideal compressed cache, the effective capacity should be the same as average compression ratio for each benchmark. However, low memory intensive workloads with even good compressibility, such as ammp, do not have that large working set. That is why even when doubling the cache size (2X BASELINE) these workloads cannot use the whole cache (i.e., average normalized capacity of 1.6 with 2X BASELINE).

By packing compressed blocks, YACC improves effective capacity by up to 3.5 times for mix3 and on average by 84%. YACC on average improves cache effective capacity similar to a 2X BASELINE, while it has almost half area. Among our workloads, high memory intensive workloads benefit the most from YACC, and in general compression. YACC increases cache capacity more than twice for these workloads.

Although similar to SCC, YACC compacts neighboring blocks with similar compressibility, it improves effective capacity over YACC. YACC proposes in-place expansion, avoiding extra replacements when a block is the only resident in one data entry. It also packs two non-contiguous neighbors (like M and P in Figure 3) with CF of two in one entry. Thus, due to these optimizations, YACC achieves better effective capacity over SCC.

Among previous work, DCC provides higher normalized effective capacity than SCC and YACC. In SCC and YACC, only neighbors with similar compressibility share the space. In DCC, however, blocks can be stored anywhere in the cache, so the space freed by compressing one block can be used to store a non-neighboring block. Thus, overall, DCC provides the highest effective capacity, but at the cost of a more than four times the area overhead than YACC and SCC and more complex data access path.

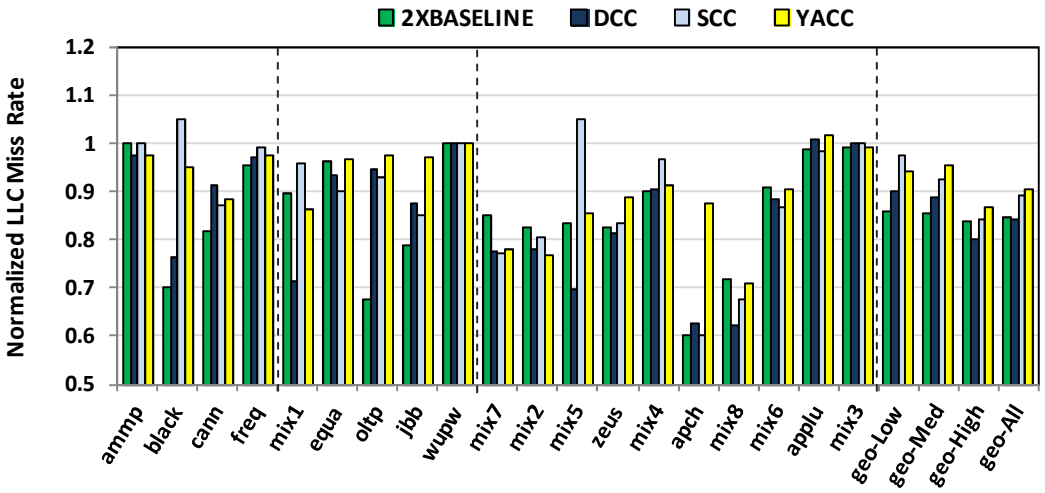


Fig. 8. Normalized LLC miss rate (MPKI). Applications are ordered based on their Baseline MPKI.

5.3 Cache Miss Rate

By improving cache effective capacity, compressed caches tend to reduce cache miss rate. Figure 8 shows the LLC MPKI (Misses per Kilo executed Instructions) for different cache designs. When doubling the cache size, 2X Baseline improves LLC MPKI by on average 15%, and up to 40% for apache. However, these benefits come at twice larger LLC area, which is already one of the largest on-chip components.

YACC improves LLC MPKI by compressing blocks. It achieves most of the benefits of 2x Baseline, with about half area. YACC improves LLC MPKI by about 10% on average, and up to 30%. Among previous works, YACC performs similar to SCC. SCC uses compression, but limits cache effective associativity by only mapping a block into 4 out of 16 cache ways. On the other hand, it employs skewing to compensate for possible loss of associativity. Thus, for some workloads, such as apache, skewing combined with compression can improve overall miss rate, achieving lower LLC MPKI than YACC. While for some others, such as mix5, skewing would not compensate lower associativity in SCC. Compared to DCC, DCC lowers MPKI

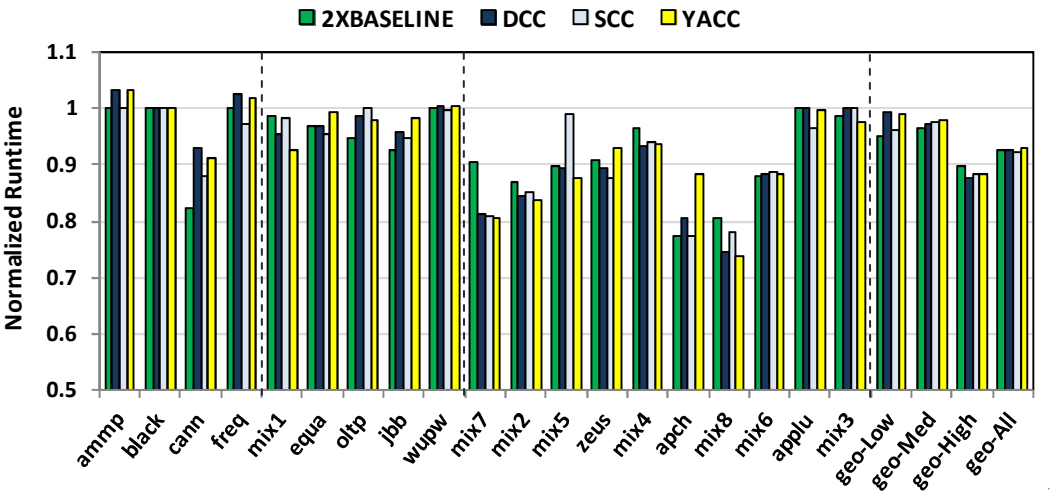


Fig. 9. Normalized performance

more than YACC and SCC, but at higher design complexity and overheads. Also note that on a few applications (applu, mix3, freq), doubling the cache size has virtually no impact on the miss rate, therefore a large compression factor on mix3 does not significantly help to reduce the misprediction rate.

5.4 System Performance and Energy

By improving cache effective capacity and lowering cache miss rate, compressed caches can improve system performance and energy. Figure 9 and Figure 10 show system performance and energy of different cache designs. We report total energy of cores, caches, on-chip network, and main memory, including both leakage and dynamic energy.

YACC improves performance by up to 26%, and on average 8%. It similarly lowers total energy by up to 20% and on average 6% by reducing the number of accesses to main memory and lowering the runtime. Overall, cache sensitive applications, for which miss rate reduces when increasing cache size, benefit the most from compression. Among our evaluated workloads, many applications from medium and high memory intensive category, such as jbb, apache, and mix8, benefit the most from YACC. On the other hand, cache insensitive workloads, which include low memory intensive workloads as well as some with very high miss rate would not benefit from YACC. For example, libquantum (mix3), which has about 43 MPKI, does not benefit from compression despite its high compressibility.

YACC achieves similar performance and energy benefits as 2x Baseline, and previous works, SCC and DCC, with lower design complexity and overheads. In general, memory intensive workloads benefit the most from larger cache capacity and compression. YACC achieves on average 12% shorter runtime for high memory intensive workloads, such as apache and zeus. On the other hand, it achieves the lowest benefit (on average 2% better performance) for low and medium memory intensive workloads, such as equake and wupwise.

Figure 11 (b) shows how energy breakdown changes over Baseline in YACC. For example, in ammp, using compression would hurt the performance for about 3%. This would cause a similar increase in static energy of system. In addition to add, because of compression and decompression, we will have about 1% higher dynamic energy in caches. Overall, we see about 4% higher energy usage for ammp. On the other hand, for mix8, where we have about 26% lower runtime, we see an overall of 19% lower energy. For this application, in addition to saving static energy, the majority of

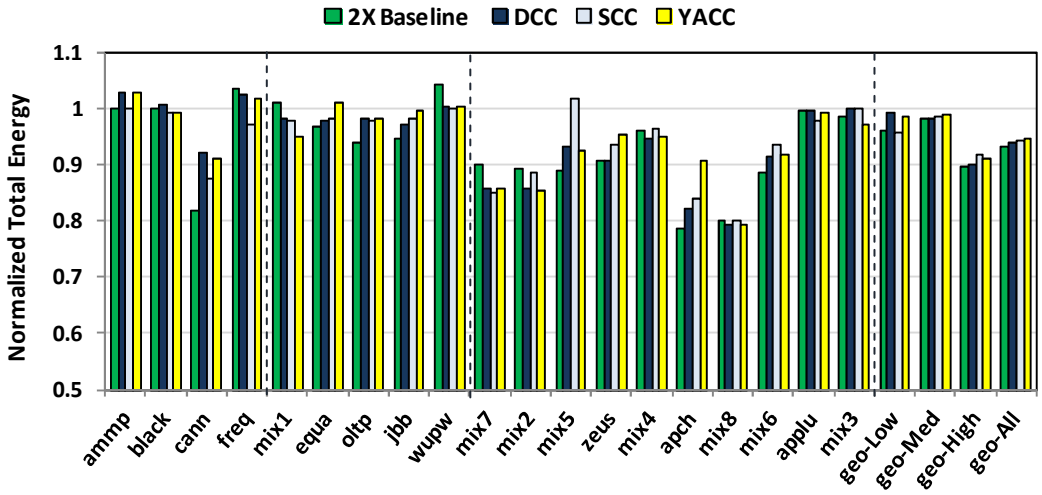


Fig. 10. Normalized total energy

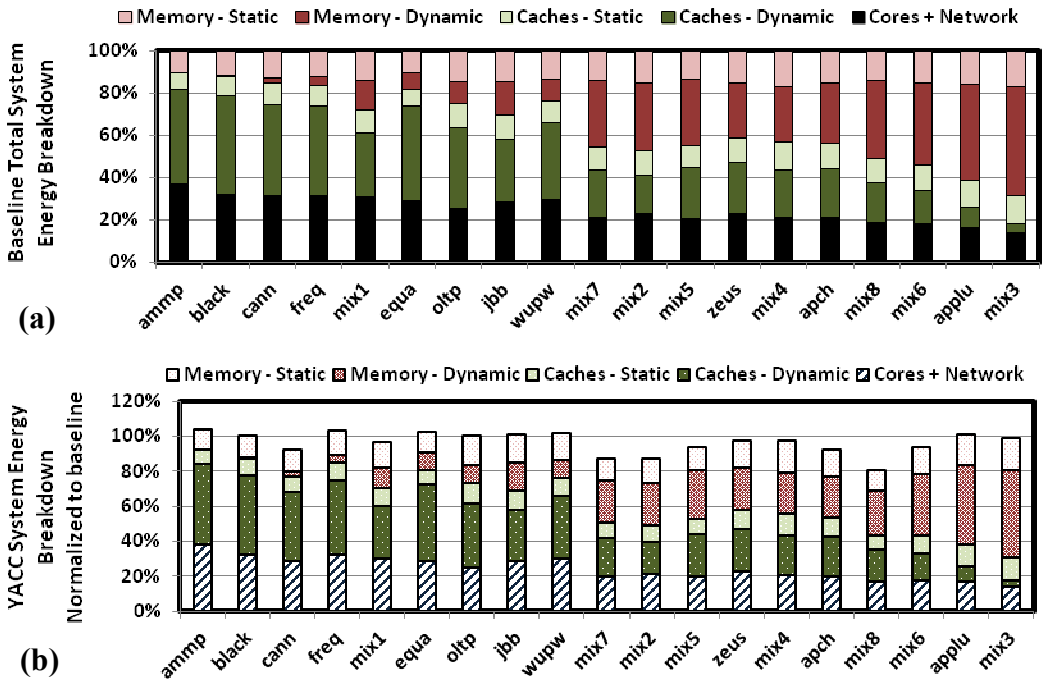


Fig. 11.Energy breakdown of Baseline configuration (a). Energy breakdown of YACC normalized to Baseline configuration (b).

benefit (11% out of 19%) comes from lowering dynamic energy of off-chip memory by reducing L3 cache miss rate.

YACC achieves similar performance and energy benefits as DCC and SCC, but YACC has a simpler design that can easily employ any replacement policy. In Figure 13, we illustrate the performance of YACC when using RRIP replacement policy [Jaleel et al. 2010]. For this experiment we use static RRIP, and use 2 bits per super-block tag to store 4 possible re-reference prediction values (RRPV). When allocating a super-block tag for the first time, we set its RRPV to '2' (3-1). On a hit, we promote that super-block tag by setting its RRPV to '0'. When replacing, YACC-RRIP would pick the super-block tag with RRPV of '3'. We also experimented YACC-RRIP with 3-bit RRPV (8 levels) as well. In addition, we considered promoting a super-block tag when inserting a block to that data entry. However, those configurations performed similar to what we presented here. For Baseline-RRIP, we use a similar configuration. We use 2-bit RRPV per block, and promote on cache hits.

As shown in Figure 12, when using RRIP, on average YACC (YACC-RRIP) performs similar to using LRU (YACC-LRU). A regular uncompressed cache with RRIP replacement policy (Baseline-RRIP) also performs on average similar to Baseline-LRU, improving performance for some workloads (e.g., zeus), while lowering or not impacting performance for others (e.g., mix1 and ammp). Overall, this experiment shows that YACC can use alternative replacement policies, while providing its benefits from compression.

6. CONCLUSIONS

In the few past years, several proposals have addressed many issues preventing the effective hardware implementation of compressed caches. Our previous proposals, DCC and SCC, reduce the extra hardware complexity induced for storing and retrieving compressed data blocks. In practice with SCC [Sardashti et al. 2014], we

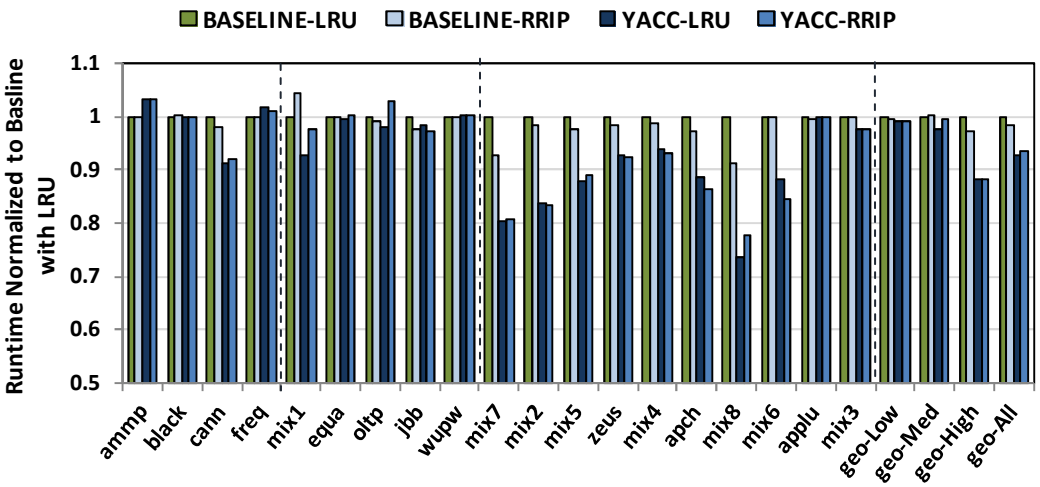


Fig. 12. YACC performance using RRIP

addressed most of the issues of the compaction in caches: very limited tag and metadata overhead, direct tag-data matching, no need for defragmentation. However, this comes at the cost of using skewing that induces using one decoder per tag way and has not been widely adopted by industry.

In this paper, we introduce YACC that is a simple hardware compressed cache design achieving the high benefits of previous proposals while significantly simplifies the design. We show that YACC achieves performance and energy benefits comparable to that of a conventional cache with twice the capacity, and previous work DCC and SCC. However, YACC does this with lower complexity (no skewing) and very limited storage overheads (only 8 extra tag bits per 64 bytes of storage).

REFERENCES

A. Alameldeen, and D. Wood 2004. Adaptive Cache Compression for High-Performance Processors. In Proceedings of the 31st Annual International Symposium on Computer Architecture, 2004.

A. Seznec. 1993. A case for two-way skewed-associative caches. In Proc. of the 20th annual Intl. Symp. on Computer Architecture, 1993.

A. Seznec. 2004. Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB. IEEE Transactions on Computers, 2004.

A. Seznec, F. Bodin. 1993. Skewed-associative caches. Proceedings of PARLE' 93, Munich, June 1993, also available as INRIA Research Report 1655. http://hal.inria.fr/docs/00/07/49/02/PDF/RR-1655.pdf.

Alaa R. Alameldeen, Milo M. K. Martin, Carl J. Mauer, Kevin E. Moore, Min Xu, Mark D. Hill, David A. Wood, Daniel J. Sorin. 2003. Simulating a $2M Commercial Server on a $2K PC, IEEE Computer, 2003.

A. Alameldeen, D. Wood 2003. Variability in Architectural Simulations of Multi-threaded Workloads, In Proceedings of the Ninth IEEE Symposium on High-Performance Computer Architecture, 2003.

Andre Seznec. 1994. "Decoupled sectored caches: Conciliating low tag implementation cost and low miss ratio," International Symposium on Computer Architecture, 1994.

Angelos Arelakis, Per Stenstrom. 2014. SC2: A statistical compression cache scheme, In Proceedings of the 41st Annual International Symposium on Computer Architecture, 2014.

Bulent Abali, Hubertus Franke, Xiaowei Shen, Dan E. Poff, and T. Basil Smith. 2001. Performance of Hardware Compressed Main Memory, In Proceedings of the 7th IEEE Symposium on High-Performance Computer Architecture, 2001.

CACTI: http://www.hpl.hp.com/research/cacti/

Christian Bienia and Kai Li. 2009. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors, In Workshop on Modeling, Benchmarking and Simulation, 2009.

Daniel Sanchez, Christos Kozyrakis. 2010. The ZCache: Decoupling Ways and Associativity, in Proceedings of the 43rd annual IEEE/ACM international symposium on Microarchitecture, 2010.

E. Hallnor, S. Reinhardt. 2005. A Unified Compressed Memory Hierarchy, In Proceedings of the 11th International Symposium on High-Performance Computer Architecture, 2005.

Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, Todd C. Mowry. 2013. Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework, Annual IEEE/ACM International Symposium on Microarchitecture, 2013.

Intel Core i7 Processors: http://www.intel.com/products/processor/corei7/

Jang-Soo Lee, Won-Kee Hong, Shin-Dug Kim. 2000. An on-chip cache compression technique to reduce decompression overhead and design complexity, Journal of Systems Architecture, 2000.

Jason Zebchuk, Elham Safi, and Andreas Moshovos. 2007. A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy, In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007.

Jeffrey Rothman and Alan Smith. 1999. The Pool of Subsectors Cache Design, International Conference on Supercomputing, 1999.

Julien Dusser, Andre Seznec. 2011. Decoupled Zero-Compressed Memory, In Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, 2011.

Julien Dusser, Thomas Piquet, André Seznec. 2009. Zero-content augmented caches, In Proceedings of the 23rd international conference on Supercomputing, 2009.

Jun Yang and Rajiv Gupta. 2002. Frequent Value Locality and its Applications, ACM Transactions on Embedded Computing Systems, 2002.

Luis Villa, Michael Zhang, and Krste Asanovic. 2000. Dynamic zero compression for cache energy reduction, In Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture, 2000.

M. Ekman, P. Stenstrom. 2005. A robust main-memory compression scheme, SIGARCH Computer Architecture News, 2005.

M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood 2005. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset, Computer Architecture News, 2005.

Nam Sung Kim, Todd Austin, Trevor Mudge. 2002. Low-Energy Data Cache Using Sign Compression and Cache Line Bisection, Second Annual workshop on Memory Performance Issues, 2002.

Somayeh Sardashti and David A. Wood 2013. Decoupled Compressed Cache: Exploiting Spatial Locality for Energy-Optimized Compressed Caching, Annual IEEE/ACM International Symposium on Microarchitecture, 2013.

Seungcheol Baek, Hyung Gyu Lee, Chrysostomos Nicopoulos, Junghee Lee, and Jongman Kim. 2013. ECM: Effective Capacity Maximizer for High-Performance Compressed Caching, In Proceedings of IEEE Symposium on High-Performance Computer Architecture, 2013.

Somayeh Sardashti and David A. Wood 2013. Decoupled Compressed Cache: Exploiting Spatial Locality for Energy Optimization, in IEEE Micro Top Picks from the 2013 Computer Architecture Conferences.

S. Sardashti, A. Seznec, and D. Wood 2014. Skewed Compressed Caches, in the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47), 2014.

Soontae Kim, Jesung Kim, Jongmin Lee, Seokin Hong. 2011. Residue Cache: A Low-Energy Low-Area L2 Cache Architecture via Compression and Partial Hits, In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, 2011.

Vishal Aslot, M. Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. 2001. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance, In Workshop on OpenMP Applications and Tools, 2001.

Xi Chen, Lei Yang, Robert P. Dick, Li Shang, and Haris Lekatsas. 2010. C-pack: a high-performance microprocessor cache compression algorithm, IEEE Transactions on VLSI Systems, 2010.

Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., Joel Emer. 2010. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP), ISCA 2010.

Jacob Ziv and Abraham Lempel. 1977. A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory, 23(3):337–343, May 1977.

Jacob Ziv and Abraham Lempel. 1978. Compression of Individual Sequences Via Variable-Rate Coding. IEEE Transactions on Information Theory, 24(5):530 –536, September 1978.

David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. Proc. Inst. Radio Engineers, 40(9):1098–1101, September 1952.

Jeffrey Scott Vitter. 1987. Design and Analysis of Dynamic Huffman Codes. Journal of the ACM, 34(4):825–845, October 1987.

Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. Base-delta-immediate compression: practical data compression for on-chip caches. In Proceedings of the 21st international conference on Parallel architectures and compilation techniques (PACT '12). ACM, New York, NY, USA, 377-388.

Calculating memory system power for DDR3. Technical Report TN 41-01. Micron Technology. 2007.